

Computer Vision  
and Geometry Lab

# Informatik I for D-MAVT

## Exercise Session 10

# Organisatorisches

- Teaching Assistant
  - Alexander Schwing(aschwing@inf.ethz.ch)
  - CAB G 89
  - Website:  
<http://www.alexander-schwing.de/Info1DMaVt/>
- Übungsabgabe
  - Auf Papier, keine Emails
    - Später wenn Programme umfangreicher werden, ev. auch Abgabe per Email möglich
- Testatbedingungen
  - 75% aller Übungen
    - Voraussichtlich 12 Serien (d.h. 9 gelöste Serien)

# Themen

- Nachbesprechung
- Repetition (Klassen, Konstruktoren)
- Operatoren
- Friends
- Templates
- Vorbesprechung Serie 10

# Nachbesprechung

- Taschenrechner:
  - Rchte Reihenfolge der Operatoren nach pop
- Stack:
  - siehe Repetition

# Repetition

- Eine Klasse ist eine Datenstruktur mit Funktionen
- Ein Objekt ist eine Instanz einer Klasse
- Die Zugriffsart (access specifiers) zeigt an wer auf welche Attribute/Methoden zugreifen darf
  - Private (default)
  - Public
  - Protected

# Repetition

- Konstruktoren
  - Werden beim Erzeugen aufgerufen und sind verantwortlich für korrekte Initialisierung
- Destruktoren
  - Werden beim Auflösen eines Objekts aufgerufen und sind verantwortlich für Freigabe des Speichers
- Konstruktoren können nicht wie normale Funktionen aufgerufen werden!

# Repetition

- Default Konstruktoren
  - Default C'tor: `MyClass::MyClass()`
    - Selbstinitialisierung
  - Default Copy C'tor: `MyClass::MyClass(const MyClass& )`
    - Initialisierung von einer anderen Klasse des selben Typs

```
MyClass c = a + b; MyClass d(c);
```
  - Default Assignment Operator: `MyClass&`  
`MyClass::operator=(const MyClass&)`  
`c = a + b;`
- Sobald ein eigener C'tor definiert wird gibt es keinen Default C'tor mehr

# Repetition

## ■ Beispiel

```
#include <memory>

class Stack{
    double *s;
    int pos;
    int cap;
public:
    Stack(int cap = 10);
    Stack(const Stack&); ←
    ~Stack();
    Stack& operator= (const Stack&); ←
};

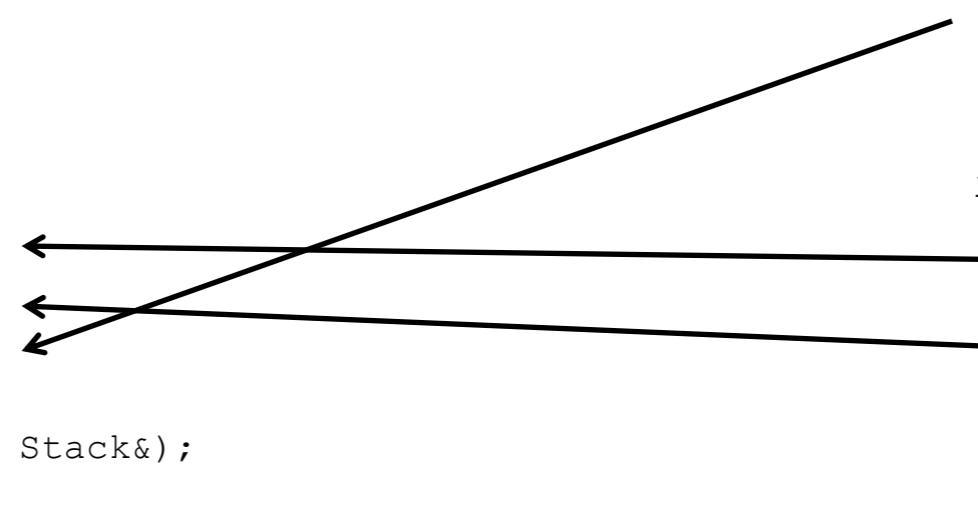
Stack::Stack(int cap) {s = new double[cap];this->cap = cap; pos = 0;}
Stack::Stack(const Stack& oc) {    s = new double[oc.cap];memcpy(s, oc.s, oc.cap*sizeof(double));
    cap = oc.cap;pos = oc.pos;}
Stack::~Stack() {delete [] s;}
Stack& Stack::operator= (const Stack& oc) {if(this!=&oc){this->~Stack();s = new double[oc.cap];
    memcpy(s, oc.s, oc.cap*sizeof(double));cap = oc.cap;pos = oc.pos;}
    return *this;
}

int main() {
    Stack a;
    Stack b(5);
    Stack c(b);
    Stack d = a;
    d = c;
    return 0;
}
```

# Repetition

## ■ Parameterübergabe By-Value

```
#include <memory>
class Stack{
    double *s;
    int pos;
    int cap;
public:
    Stack(int cap = 10);
    Stack(Stack& oc);
    ~Stack();
    Stack& operator= (const Stack&);
};
```

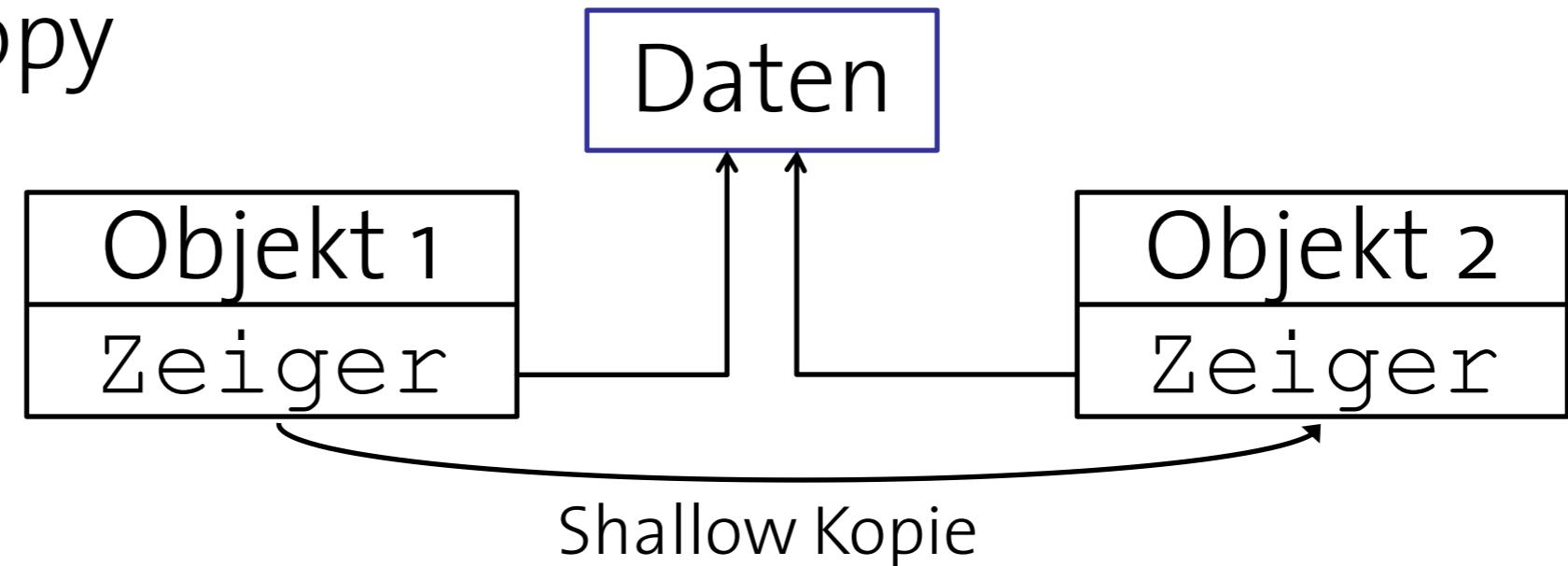


```
void doSomething(Stack tmp) {
    ...
}
int main() {
    Stack a;
    doSomething(a);
    return 0;
}
```

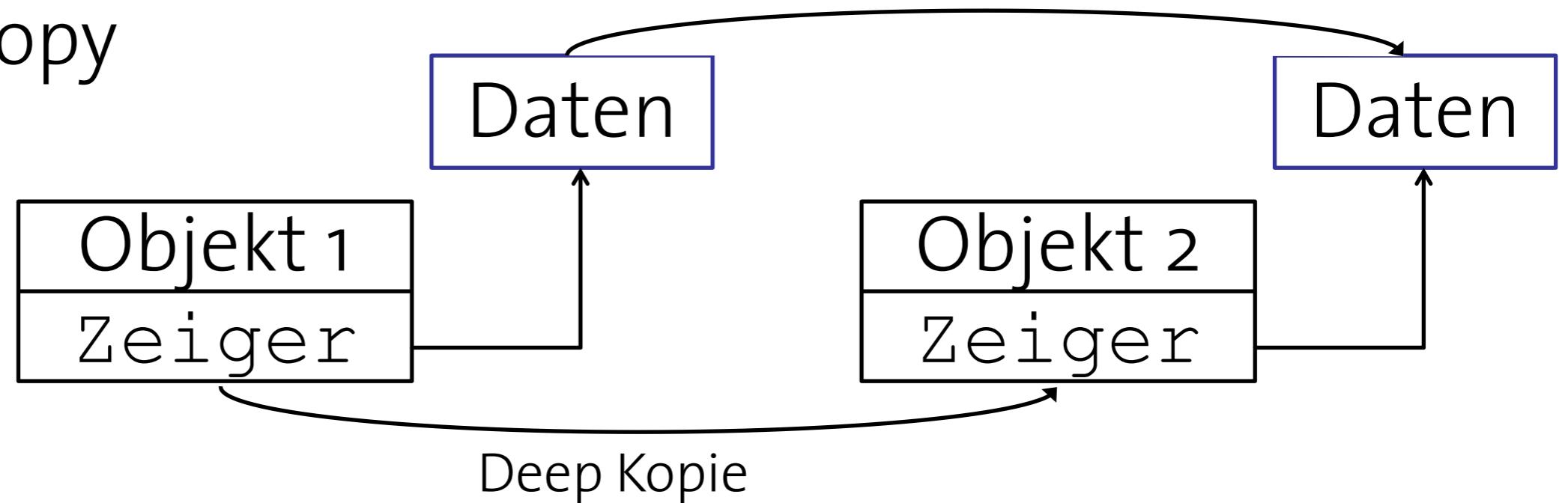
## ■ Vorsicht bei Klassen die dynamische Speicherallokation verwenden (→ eigene Konstruktoren und Operatoren für deep copy)

# Repetition

- Shallow copy



- Deep copy



# Repetition

## ■ Konstruktor:

```
Stack::Stack(int cap = 10) {  
    s = new double[cap];  
    this->cap = cap;  
    pos = 0;  
}
```

## ■ Copy-Konstruktor:

```
Stack::Stack(const Stack& oc) {  
    s = new double[oc.cap];  
    memcpy(s, oc.s, oc.cap*sizeof(double));  
    cap = oc.cap; pos = oc.pos; }
```

## ■ Destruktor:

```
Stack::~Stack() {  
    delete [] s;  
}
```

## ■ Assignment Operator:

```
Stack& Stack::operator= (const Stack& oc) {  
    if(this!=&oc) {  
        this->~Stack();  
        s = new double[oc.cap];  
        memcpy(s, oc.s, oc.cap*sizeof(double));  
        cap = oc.cap; pos = oc.pos;  
    }  
    return *this;
```

# Operatoren

- Spezielle **Funktionen** die mit Operatoren (+,-,\*,/,...) aufgerufen werden
- Syntax:  
`Rückgabetyp operator Zeichen ( Parameterliste ) { Block }`
- Operatorfunktion kann überladen werden
- Operatorfunktionen müssen nicht zu einer Klasse gehören (können auch globale definiert sein)
- Operatorfunktionen haben „erprobte“ Syntax, d.h. der Funktionskopf sieht fast immer ähnlich aus

# Operatoren

## ■ Oft auftretende Syntax:

Operator name	Syntax	Overloadable	Include d <a href="#">in C</a>	Prototype examples (T1 and T2 are types)	
				As member of T1	Outside class definitions
<a href="#">Unary plus</a>	+a	Yes	Yes	T1 T1::operator +() const;	T1 operator +(const T1& a);
<a href="#">Addition</a>	a + b	Yes	Yes	T1 T1::operator +(const T2& b) const;	T1 operator +(const T1& a, const T2& b);
<a href="#">Increment</a>	Prefix	++a	Yes	T1& T1::operator ++();	T1& operator ++(T1& a);
	Postfix	a++	Yes	T1 T1::operator ++(int);	T1 operator ++(T1& a, int);
Assignment by addition	a += b	Yes	Yes	T1& T1::operator +=(const T2& b);	T1& operator +=(T1& a, const T2& b);
Unary minus (negation)	-a	Yes	Yes	T1 T1::operator -() const;	T1 operator -(const T1& a);
<a href="#">Subtraction (difference)</a>	a - b	Yes	Yes	T1 T1::operator -(const T2& b) const;	T1 operator -(const T1& a, const T2& b);
<a href="#">Decrement</a>	Prefix	--a	Yes	T1& T1::operator --();	T1& operator --(T1& a);
	Postfix	a--	Yes	T1 T1::operator --(int);	T1 operator --(T1& a, int);
Assignment by subtraction	a -= b	Yes	Yes	T1& T1::operator -=(const T2& b);	T1& operator -=(T1& a, const T2& b);
<a href="#">Multiplication (product)</a>	a * b	Yes	Yes	T1 T1::operator *(const T2& b) const;	T1 operator *(const T1 &a, const T2& b);

Wikipedia

# Operatoren

```
class CRational {  
    int numerator, denominator;  
public:  
    CRational(const int, const int);  
    const CRational operator+(const CRational&) const;  
    const CRational operator+(const int n) const;  
};  
  
CRational::CRational(const int nNum, const int nDenom) {  
    numerator = nNum;  
    denominator = nDenom;  
}  
  
const CRational CRational::operator+(const int n) const {  
    CRational var(this->numerator + n*this->denominator, this->denominator);  
    return var;  
}  
  
const CRational CRational::operator +(const CRational &r) const {  
    CRational var(this->numerator*r.denominator + r.numerator*this->denominator,  
r.denominator*this->denominator);  
    return var;  
}
```

## Outside Class Definition

```
CRational operator+(int& n, CRational& r) {  
    return r+n;  
}  
  
int main() {  
    CRational half(1,2);  
    CRational tmp = half + 2;  
    CRational tmp1 = half + tmp;  
    CRational tmp2 = 2 + tmp1;  
    return 0;  
}
```

## Inside Class Definition

# Friends

- innerhalb der Definition einer Klasse können **Funktionen** oder **andere Klassen** angegeben werden, die auf die privaten Daten der Klasse zugreifen dürfen.
- Andere Klasse: `friend class OtherClassName`
- Nur eine Funktion einer anderen Klasse:  
`friend ReturnType OtherClassName::otherMethod (...)`
- Globale Funktion:  
`friend ReturnType FunctionName (...)`

# Friends

- Friends führen zu intuitiverem Code und sind nötig um Operator Overloading korrekt (effizienter) zu implementieren
- *“Friend functions do not break encapsulation; instead they naturally extend the encapsulation barrier”*

# Friends

## ■ Beispiel 1:

```
#include <iostream>
using namespace std;

class CRational {

    int numerator, denominator;

    friend ostream& operator<<(ostream &out, const CRational &r);

public:

    CRational(const int nNumerator, const int nDenominator);

    CRational::CRational(const int nNumerator, const int nDenominator) :
        numerator(nNumerator), denominator(nDenominator) {}

    ostream& operator<<(ostream& out, const CRational &r) {

        // Access private members of CRational to display r.
        out << r.numerator << "/" << r.denominator;

        return out;
    }

    int main() {

        CRational half(1, 2);

        cout << half << endl;

        return 0;
    }
}
```

# Friends

## ■ Beispiel 2:

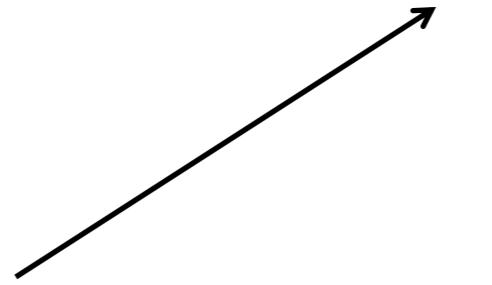
```
class CRational {  
    int numerator, denominator;  
public:  
    CRational(const int, const int);  
    void divide(int);  
};  
  
CRational::CRational(const int nNum, const int nDenom) :  
    numerator(nNum), denominator(nDenom) {}  
  
void CRational::divide(int n) {  
    denominator *= n;  
}  
  
int main() {  
    CRational half(1, 2);  
    half.divide(2);  
    return 0;  
}
```

```
class CRational {  
    int numerator, denominator;  
    friend CRational operator/(CRational& r, int n);  
public:  
    CRational(const int, const int);  
};  
  
CRational::CRational(const int nNum, const int nDenom) :  
    numerator(nNum), denominator(nDenom) {}  
  
CRational operator/(const CRational& r, const int n) {  
    CRational out(r.numerator, r.denominator*n);  
    return out;  
}  
  
int main() {  
    CRational half(1, 2);  
    CRational quater = half/2;  
    return 0;  
}
```

# Friends

## ■ Beispiel 3

```
class CRational {  
    int numerator, denominator;  
    friend CRational operator+(int, CRational&);  
public:  
    CRational(const int, const int);  
    const CRational operator+(const CRational&) const;  
    const CRational operator+(const int n) const;  
};  
  
CRational::CRational(const int nNum, const int nDenom) {  
    numerator = nNum;  
    denominator = nDenom;  
}  
  
const CRational CRational::operator+(const int n) const {  
    CRational var(this->numerator + n*this->denominator, this->denominator);  
    return var;  
}  
  
const CRational CRational::operator +(const CRational &r) const {  
    CRational var(this->numerator*r.denominator + r.numerator*this->denominator, r.denominator*this->denominator);  
    return var;  
}
```



```
CRational operator+(int n, CRational& r) {  
    return r+n;  
}  
  
int main() {  
    CRational half(1,2);  
    CRational tmp = half + 2;  
    CRational tmp1 = half + tmp;  
    CRational tmp2 = 2 + tmp1;  
    return 0;  
}
```

# Templates

- Werkzeug um einen Algorithmus allgemein zu beschreiben ohne den Datentyp festzulegen
- Compiled “on demand”, d.h. nur bei Verwendung
- Algorithmen sind in der Regel unabhängig vom Datentyp (int, float, double, ...)
  - Minimum/Maximum/Swap
  - Stack
  - Sortieren
  - ...

# Templates

## ■ Beispiele

```
double min(double a, double b) {  
    return a<b ? a : b;  
}
```

```
double max(double a, double b) {  
    return a>b ? a : b;  
}
```

```
void swap(double& a, double& b) {  
    double tmp = a;  
    a = b;  
    b = tmp;  
}
```

```
template <class T>  
T min(T a, T b) {  
    return a<b ? a : b;  
}
```

```
template <class T>  
T max(T a, T b) {  
    return a>b ? a : b;  
}
```

```
template<class T>  
void swap(T& a, T& b) {  
    T tmp = a;  
    a = b;  
    b = tmp;  
}
```

# Templates

```
#include <iostream>

int maxIX(double* arr, int len) {
    int mI = 0;
    for(int i=1;i<len;++i) {
        if(arr[i]>arr[mI])
            mI = i;
    } return mI;
}

void maxSort(double* arr, int n) {
    if (n > 1) {      int m = maxIX(arr, n);
        double tmp = arr[m];
        arr[m] = arr[n-1];
        arr[n-1] = tmp;
        maxSort(arr, n-1);
    }
}

int main() {
    double values[] = {9, 8, 7, 6, 5, 4, 3, 2, 1};
    maxSort(values, 9);
    for(int k=0;k<9;++k) std::cout << values[k];
    return 0;
}

#include <iostream>
template <class T>
int maxIX(T* arr, int len) {
    int mI = 0;
    for(int i=1;i<len;++i) {
        if(arr[i]>arr[mI])
            mI = i;
    } return mI;
}

template <class T>
void maxSort(T* arr, int n) {
    if (n > 1) {      int m = maxIX(arr, n);
        T tmp = arr[m];
        arr[m] = arr[n-1];
        arr[n-1] = tmp;
        maxSort(arr, n-1);
    }
}

int main() {
    double values[] = {9, 8, 7, 6, 5, 4, 3, 2, 1};
    maxSort(values, 9);
    for(int k=0;k<9;++k) std::cout << values[k];
    return 0;
}
```

# Templates

## ■ Beispiel: Templates für Klassen

```
template <class T>
class mypair {
    T values [2];
public:
    mypair (T first, T second) {
        values[0]=first; values[1]=second;
    }
};

int main() {
    mypair<int> Instance1(1, 2/3);           // (1, 0)
    mypair<int> Instance2(1, -1.7);          // (1, -1)
    mypair<double> Instance3(1, 2/3);         // (1.0, 0.0)
    mypair<double> Instance4(1, -1.7);        // (1.0, -1.7)
    return 0;
}
```

# Templates

## ■ Non-Type Parameters

```
#include <iostream>
using namespace std;
template <class T = char, int N = 10>
class mysequence {
    T memblock [N];
public:
    void setmember (int x, T value);
    T getmember (int x);
};
template <class T, int N>
void mysequence<T,N>::setmember (int x, T value) {
    memblock[x]=value;
}
template <class T, int N>
T mysequence<T,N>::getmember (int x) {
    return memblock[x];
}
int main () {
    mysequence <int,5> myints;
    mysequence <double,5> myfloats;
    mysequence <> mychars;
    myints.setmember (0,100);
    myfloats.setmember (3,3.1416);
    cout << myints.getmember(0) << '\n';
    cout << myfloats.getmember(3) << '\n';
    return 0;
}
```

# Vorbesprechung Serie 10

- Programmierübung: Klasse Polynom
  - Dynamisches Array mit Polynomkoeffiziente
  - → Operatoren selbst schreiben

```
class cPolynom
{
private:
    double *coef;
    int deg;
public:
    // Ausgabefunktion fuer das Polynom
    void print() const;
};
```

# Vorbesprechung Serie 10

- A) Konstruktoren
  - `cPolynom(double* coef, int degree);`
  - `cPolynom(int degree);`
- B) Destruktor
  - `~cPolynom();`
- C) Eval Methode
- D) Addition
  - `const cPolynom operator+(const cPolynom& r) const;`
- E) Subtraktion (siehe Addition)

# Vorbesprechung Serie 10

- F) Kopierkonstruktor
  - `cPolynom(const cPolynom& r);`
- G) Assignment Operator
  - `cPolynom& operator=(const cPolynom& r);`